National Research Council Canada
Conseil national de recherches Canada

Division of Electrical Engineering
Division de génie électrique

ERB-1006

Prismatic-shaped Block Insertion: An Application of a Multiprocessor Robot Controller using Harmony

D.H. O'Hara and R. Kurtz

December 1987

Canada

NRCC No. 28493

88-04448/

Copyright 1987 by
National Research Council of Canada

Permission is granted to quote short excerpts and to reproduce figures and tables from this report, provided that the source of such material is fully acknowledged.

Copyright 1987 par
Conseil national de recherches du Canada

Il est permis de citer de courts extraits et de reproduire des figures ou tableaux du présent rapport, à condition d'en identifier clairement la source.

Additional copies are available free of charge from:

Editorial Office, Room 301
Division of Electrical Engineering
National Research Council of Canada
Ottawa, Ontario, Canada
K1A 0R8

Des exemplaires supplémentaires gratuits peuvent être obtenus à l'adresse suivante:

Bureau des publications, Pièce 301
Division de génie électrique
Conseil national de recherches du Canada
Ottawa (Ontario) Canada
K1A 0R8

PRISMATIC-SHAPED BLOCK INSERTION: AN APPLICATION OF A MULTIPROCESSOR ROBOT CONTROLLER USING HARMONY

D. H. O'Hara and R. Kurtz

ERB-1006
NRCC No. 28493

December 1987

TABLE OF CONTENTS

	Page
ABSTRACT/RÉSUMÉ	v
INTRODUCTION	1
APPLICATION OF HARMONY	2
BLOCK INSERTION PROCEDURE	4
Block Description	4
Description of Robot Frames of Reference	5
Insertion Procedure	6
Initial Position and Orientation	6
Motion Sequences	6
INSERTION PROGRAM ORGANIZATION	7
Introduction	7
Task Organization and Descriptions	8
Alt_handler(): Controlling Robot Motions from Harmony using ALTER	10
SUMMARY	14
ACKNOWLEDGEMENT	14
REFERENCES	14
APPENDIX A	17
BLOCK INSERTION DEMO	21
MONITOR TASK	24
SEQUENCE TASK	30
Alt_handler	43
TASK MIDPOINTS	52
TASK AIOHANDLER	53
TASK DISPLAY	56
TASK DISP_COURIER	59

LIST OF FIGURES

1	Program layout inter-task communication	3
2	Block tool frame	4
3	Block insertion: (a) initial stage - point contact; (b) final stage - block insertion	5

ABSTRACT

This report describes an application of the Chorus/Harmony computing facility for realtime robot control. The ALTER link on the PUMA family of robots has been adapted to provide versatile motion control from an external computer. Two aspects of the Harmony Operating System are demonstrated: firstly, the use of courier tasks to provide synchronized non-blocking inter-task communication and, secondly, the use of servers to distribute resources effectively.

The block insertion task requires immediate manipulator control response. Both active accommodation through Harmony/Chorus manipulator control from the IRCC signal data and passive accommodation of the IRCC device were used in this application.

The elements described in this report form the fundamentals of a robot controller capable of integrating sensory data for responsive motion control in Cartesian space. A video recording (U-Matic or VHS) of this demonstration is available on loan.

RÉSUMÉ

Le présent rapport décrit une application de l'installation de calcul Chorus/Harmony utilisée pour la commande de robots en temps réel. La liaison ALTER de la famille de robots PUMA a été adaptée pour permettre une commande polyvalente à partir d'un ordinateur extérieur. Deux aspects du système d'exploitation Harmony sont décrits : premièrement l'emploi de tâches de courrier pour permettre une communication intertâches sans groupage synchronisée et, deuxièmement, l'emploi de serveurs pour distribuer efficacement les ressources.

L'insertion des blocs requiert une réponse immédiate de la commande du manipulateur. Dans la présente application, une prise en charge active par l'intermédiaire de la commande du manipulateur Harmony/Chorus à partir des signaux IRCC a été utilisée conjointement avec une prise en charge passive du dispositif IRCC.

Les éléments décrits dans le présent rapport forment la base d'un contrôleur de robot capable d'intégrer des données sensorielles pour assurer une commande sensible du mouvement dans un espace cartésien. Un enregistrement vidéo (U-Matic ou VHS) de cette démonstration peut être prêté aux personnes intéressées.

PRISMATIC-SHAPED BLOCK INSERTION: AN APPLICATION OF A MULTIPROCESSOR ROBOT CONTROLLER USING HARMONY*

D.H. O'Hara and R. Kurtz

INTRODUCTION

This report summarizes a series of demonstrations of the control of the PUMA 560 robot from a multiprocessor system (Chorus/Harmony) (1-3) over the ALTER (4) link. The application was the insertion of a prismatic-shaped block into its tight fitting socket. Parts mating is a topic of considerable research interest and this demonstration provides an interesting solution to one relatively challenging mating task. However, that aspect of the problem is not emphasized in this report. Rather, we describe a control structure suitable for the responsive control of a manipulator to perform this job.

The sensor data for feedback control is provided by a single sensor, the Instrumented Remote Centre of Compliance (IRCC) device from the C.S. Draper Laboratory (5). This instrument provides displacement measurements in six-degrees-of-freedom, i.e., three linear displacements X, Y, and Z and the rotational displacement about each of the three axes. These measurements are related to the centre of remote compliance, which in our experiments was a point between the halves of the robot's parallel gripper. This makes analysis relatively straightforward. However, a very important characteristic of the IRCC is its passive compliance. This feature permits a greater instantaneous tolerance to errors and as well achieves motion control beyond the resolution of the manipulator.

The control problem is one of using single sensor feedback to constrain a control vector and to determine when a goal has been reached. The PUMA 560 controller is essentially by-passed and motion specification to the manipulator is generated in the external computer. We pass Cartesian motion increments in the robot's world coordinate system to the robot every 28 ms to achieve total motion generation from the external computer. This use of the ALTER link differs from that outlined in the Unimation manual. Although motion specification is made in robot world coordinates, the insertion task requires motion planning in dynamic tool coordinates, sensor coordinates, and in the socket coordinates of the block. A complete description of the control of the robot from Chorus/Harmony is given in ref. 4. A video tape (U-matic or VHS) of this demonstration is available for loan.

---

* Mark reserved for the exclusive use of Her Majesty the Queen in right of Canada by Canadian Patents and Development Ltd.

Harmony is a multitasking, multiprocessor operating system ideally suited to realtime control. The next section will elaborate on the use of two important structures in Harmony. These are the organization of message passing primitives via the use of a courier to provide a non-blocking structure for critical tasks and the synchronization of operations and also the method sharing of resources through the use of special server tasks.

APPLICATION OF HARMONY

Realtime control implies an ability to respond immediately to requests which occur randomly. The job being performed is, by definition, not predictable or there would be no need for sensor data to monitor the progress of the activity. Typical jobs appear to the program as a group of activities progressing independently of each other, demanding service asynchronously, yet at critical points requiring synchronization to achieve a goal. This desired program behaviour is achieved in Harmony through the use of tasks. A program in Harmony is composed of many tasks. A task is much like a subroutine except that it must be explicitly created and that it runs independently and in parallel with the task that created it.

Although tasks run independently and in parallel, they must communicate or synchronize occasionally. The mechanism for this in Harmony is message passing. The four primitives used for inter-task communication are _Send(), _Receive(), _Try_receive(), and _Reply(). An understanding of the blocking behaviour is important in the description of the task structure used in the insertion program. The two primitives _Send() and _Receive() are blocking. The sender task blocks from when the _Send() primitive is called until the receiver calls the corresponding _Reply() primitive, not just until the corresponding _Receive() or _Try_receive() primitive is called by the receiver. The receiver calling the _Receive() primitive blocks until a _Send() primitive has been called by a task which wishes to communicate with it. _Try_receive() and _Reply() are non-blocking. The _Try_receive() fails immediately, returning 0, if a message is not available when the receiver calls this primitive. The _Reply() primitive is a nonblocking situation for the receiver as the sender task is assumed to be blocked, waiting for a reply.

Both _Send() and _Reply() primitives transfer data to another task. By reversing the usual role of these primitives, it is possible to prevent blocking of the originator of the message in instances where this is undesirable. Suppose task A wishes to transfer some data to task B. If A follows the obvious route of using _Send() to send to B, it will be blocked until B replies. However, reverse the communications procedure and have B send to A first. When A has the data ready for B, it can transfer the data

in a reply primitive at which time B both acquires the data and is unblocked again ready to perform its function with the data just transferred.

The foregoing discussion is relevant to the understanding of the use of courier tasks. In Fig. 1, the high level control steps generated for insertion in the tasks Monitor() or Sequence() are passed to the task Alt_Handler(), which interprets them and passes commands via the courier task Alt_courier() to Alt_server(). Alt_server() in turn sends commands over the ALTER link to the PUMA controller. This structure insures that Alt_handler() does not block. The above-mentioned method of non-blocking communcation is used whereby Alt_handler() transfers data to the Alt_courier() with a _Reply(). The courier sends to Alt_handler() and blocks waiting for data.

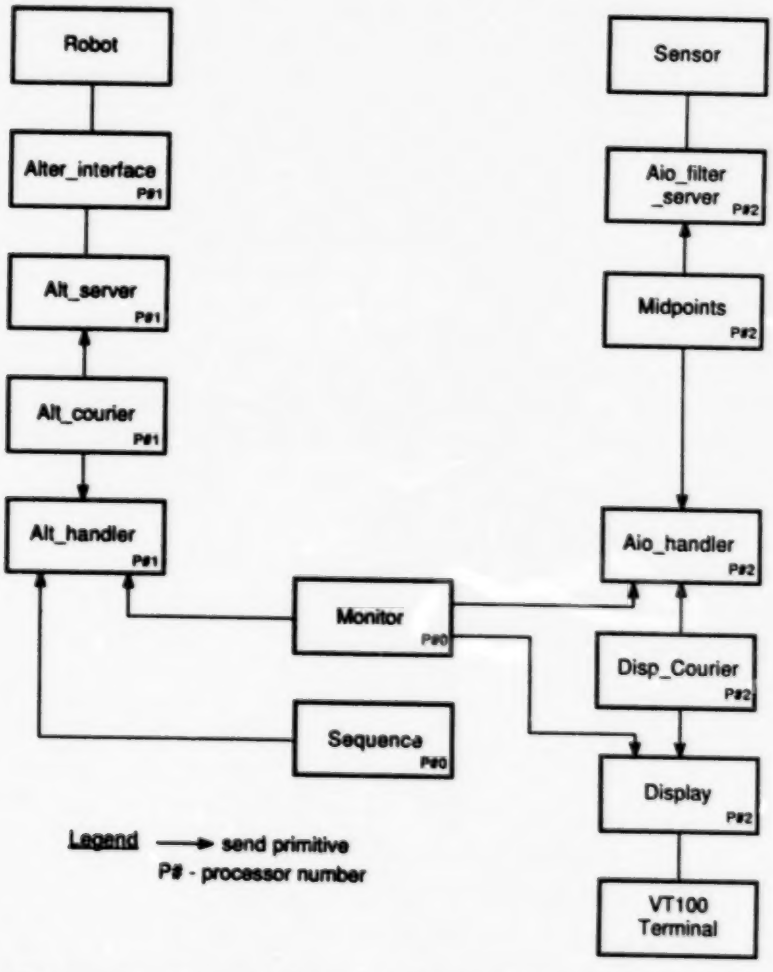


Figure 1. Program layout inter-task communication

A similar structure is used with the display task for a display of sensor data. It updates continuously the six sets of data from the IRCC as received from the Aio_handler() via a courier task Disp_courier(). The task Midpoints() acts as a courier to the Aio_handler() in the sensor data acquisition stream.

Another aspect of the Harmony capability that was exploited is the concept of a server task. This is a special type of task in Harmony which, as the name implies, provides a service to one or more clients. The insertion program uses two such tasks, Alt_handler() and Aio_handler(). Alt_handler() organizes robot motion commands and provides them to the robot controller on demand through a courier task. The higher level control tasks Sequence() and Monitor() use the services of Alt_handler() to specify robot trajectory. Aio_handler() collects sensor data and distributes the most current offset displacements to the Display() task again via a courier and to the Monitor() task. These communication paths will be described in more detail in the section on program organization. However, first, a description of the insertion algorithm will help in understanding the program requirements.

BLOCK INSERTION PROCEDURE

Block Description

The block is shown in Fig. 2 and the following description upper/lower and right/left are with reference to viewing face A of the block. The block face A is ~2 cm × 5 cm and the block is ~15 cm long. Neither socket

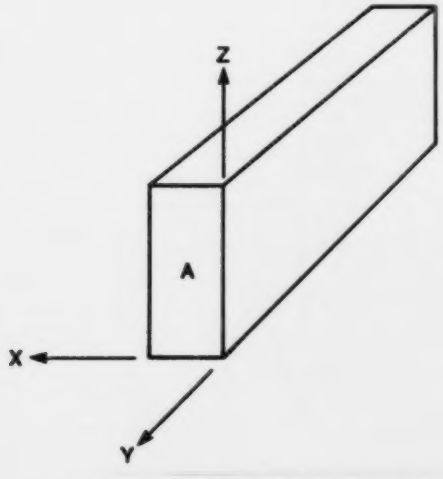
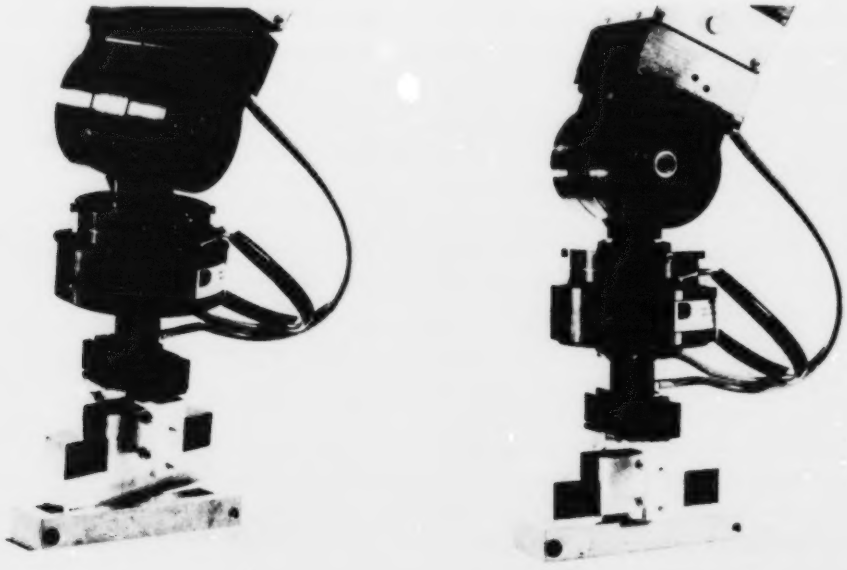


Figure 2. Block tool frame

nor block edges have chamfers. The tolerance of fit, block into socket, was not measured, but socket base requires a hole to let the air escape at a reasonable rate. Abrasive paper has been added to the block and robot gripper faces to increase resistance to slippage. Figure 3 shows the block in the robot gripper.



(a) (b)

Figure 3. Block insertion: (a) initial stage - point contact; (b) final stage - block insertion

Description of Robot Frames of Reference

This section briefly describes the transformations used in the insertion task. The first (which is used as a reference for all others) is the robot's World Coordinate Frame. This is the standard frame described in the Unimation manuals. The socket is a secondary frame defined relative to the World Frame. In general it may be any arbitrary frame. The block is moved into the socket along the socket frame Z-axis. The flange of the socket is in the X-Y plane of the socket frame. Two tool frames are used. These are the block tool frame and the IRCC tool frame. The origin of the IRCC frame is at the approximate location of the centre of remote compliance, here the midpoint of the gripper. The origin of the block frame is shown in Fig. 1, i.e., the lower right-hand corner of the A face of the block. Once calibrated, the block must not move in the gripper. The procedure for determining the tool transformations is the straightforward technique detailed in the VAL II manual. If a fixture of known pose is

used then the tool transformation for the block can be found experimentally using feedback from the IRCC to align the block in the fixture. The IRCC tool transformation is a simple Z-axis extension of the null tool transformation.

The origin of the socket frame need not be known very accurately but the plane of the flange or socket X-Y plane must be. Again the actual plane of the flange can be found experimentally using feedback from the IRCC. In tests performed by code in Appendix A, the socket X-Y plane was parallel to the robot World axis X-Y plane.

Insertion Procedure

This section describes the algorithmic procedure used to insert the block into its socket. The numbered steps correspond to step numbers in the tasks Monitor() and Sequence() in Appendix A.

Initial Position and Orientation

The block tool origin is initially placed above the right-hand flange of the socket a short distance from the front end of the socket. Precision here is not critical. The block has an initial orientation set as follows: rotation about block tool X-axis of $+\psi$, then about Y-axis of $-\theta$ and, finally, about Z-axis of $-\theta$. Again, the amount of rotation is not critical, less than 10° in these experiments, but the amount must be known. These rotations position the origin of the block tool frame adjacent to the flange.

Motion Sequences

Step 1:

From the initial position with the block above the flange move along the socket frame Z-axis to bring the block tool frame origin in contact with the socket flange.

Step 2:

Move again in the socket frame towards the socket keeping the block corner (tool origin) in contact with the flange until the corner drops over the edge of the socket.

Step 3:

Move the block into the socket along the Z-axis of the block tool frame from maintaining contact with the block vertical edge and the socket

edge until the bottom face of the block meets the opposite edge of the socket. After contact withdraw the block slightly off the flange face.

Step 4:

Rotate about tool frame Z-axis until the side face of the block touches the edge of the socket.

Step 5:

Move in the socket frame along the Y-axis until the block contacts the front edge of the socket.

Step 6:

Withdraw the block along the block tool frame Z-axis until the block is 1 mm below edge of the socket.

Step 7:

Rotate the block in the block tool frame to align the Z-axis of the block tool frame with the Z-axis of the socket frame.

Step 8:

The block now is moved down along the Z-axis in the socket frame until it reaches the bottom of the socket. No active accommodation is attempted in this step. Any misalignment is corrected through the passive compliance of the IRCC.

INSERTION PROGRAM ORGANIZATION

Introduction

The insertion program is distributed across three processors. Processor #0 performs algorithm sequencing and high level control. The tasks Monitor() and Sequence() are important here. Processor #1 is occupied with robot control. The task Alt_handler() accepts commands from the high level control tasks and interprets them for detailed motion control which is performed by Alt_server() as described in ref. 4. Processor #2 is occupied with the acquisition and preprocessing of sensor data from the IRCC. The continuous display of sensor data on a VT100 is also performed on this processor. This task layout was arrived at after some experimentation. Initial attempts to accommodate this program on two processors gave unsatisfactory performance. Integer arithmetic is used in all calculations.

The Chorus (2) multiprocessor uses Omnibyte OB668K1A single board computers which are based on the Motorola 68000 series microcomputer.

Task Organization and Descriptions

Two independent activities, data acquisition from the IRCC and robot motion control, are coordinated to execute the insertion algorithm motion sequences. A third activity, display, keeps the experimenter aware of the progress of insertion. These streams will be described, as well as their tasks and the inter-task communication that permits asynchronous processing with data transfer and task synchronization as required.

Figure 1 is a pictorial representation of the major tasks and communication links during the insertion process. Initialization and set-up processes are not shown. The number indicates the processor on which the task runs.

Three tasks on processor 2 acquire and distribute the IRCC sensor data. These are Aio_Filter_server(), Midpoints(), and Aio_handler(). Aio_filter_server() is a modified version of the Harmony Aio_server(), clock driven to read the Analog Input/Output (AIO) board every 8 ms. The datum returned by the server is the median of the previous three samples. More sophisticated filtering was not found to be required; however, other applications could insert a "filter task" at this point in the processing. Midpoints() requests values from Aio_filter_server() and applies the calibration matrix of the IRCC to produce displacement values (degrees and millimetres).

Aio_handler() accepts the sensor data from Midpoints() and organizes the distribution to client tasks. Aio_handler() sees that a client receives only the most recent data and that the client only receives them once. Clients run asynchronously and may indeed demand sensor data more quickly than they are being acquired. Aio_handler() is implemented as a server and created through the _Server_create function. Each client first opens an input stream to Aio_handler() with the Harmony_Open function. An identifying connection number is returned to the client task to be used in all subsequent requests to the Aio_handler(). Aio_handler() keeps a table of all its clients indexed by the connection number. The table contains two flags, VALID_DATA and RPLY_PENDING. A typical sequence of events will illustrate their use. A client requests data of the Aio_handler. Since VALID_DATA is false it replies to the client with the most recent data and sets VALID DATA true. The client requests data again before new data have arrived. The Aio_handler() does not reply because VALID_DATA is true and the client blocks awaiting a reply. Aio_handler() also sets the RPLY_PENDING flag. When new data arrive at the handler, all flags are

reset and waiting tasks replied to. This structure insures that the client receives the latest data, once only, and blocks upon request for data until new data are available.

The second major information stream mentioned is for robot control. Its tasks run on processor #1. Alt_server() is occupied with detailed motion control over the ALTER link. The description of this function is treated in ref. 4. The PUMA through the ALTER link imposes a 28 ms cycle. Commands must be sent through this channel at this rate or communication will be terminated by the PUMA controller.

Alt_handler() is also implemented as a server. Its task is to receive motion requests from its client tasks, and to interpret and format them in a manner that is understood by the low level control in Alt_server(). The client sends a motion request to the Alt_handler() and blocks until the motion has been completed. The request is placed on a queue to be executed by motion control software. The motion queue accepts two types of entries (see description of the MOTN_RQST struct below), queued and immediate. Queued entries are put at the end of the list and immediate entries go to the head of the line to become the next command to be executed.

The only motions accommodated by Alt_handler() are those that are described by a straight line. Each segment consists of a series of motion increments at 28 ms intervals along a vector in Cartesian space. Increments are removed from the queue as requested by the Alt_server() until all n are used and the client's motion satisifed, at which time the client is unblocked. If there are no further motions on the queue, then an "idle" is sent to satisfy the ALTER communication protocol.

The Display() task runs on processor #2. It handles two types of messages from the insertion program. Messages and sensor readings at the transition points in the sequence are written on the lower half of the VT100 screen. These messages use the _Send() primative but would be better implemented with a queue and a courier task to transfer data to Display(). Continuous updating of sensor data is written on the top of the screen using the Disp_courier() to Aio_handler() link. Disp_courier() runs at the lowest priority on its processor but still provides adequate update rate. The keyboard input is also handled through the Display() task.

These elements provide a general structure suitable to a wide range of robot control applications. We have a sensor input data stream, an output control stream, and a user interface stream. The weakness is primarily in details of the motion specification in Alt_handler() and the strategy for empty queue. Since our manipulator rates are low, a "no motion" command is acceptable. At higher rates, rapid but controlled deceleration would be preferable.

These three streams are integrated through two high level control tasks running on processor #0, Sequence() and Monitor(). Recall from the description of the algorithm that the motion specification is typically of the type "move in a specified direction until a specified sensor condition is detected." Sequence() requests motions of the type "move independently in a certain direction." Monitor() looks for the terminating condition. However, Monitor() must also maintain the constraining conditions on the Sequence() motions. A typical division of duties would be, move across a surface (Sequence()) and maintain a specified contact force on that surface (Monitor()) until an edge is detected (Monitor()). The motions sent by Sequence() are queued motions (end of the queue) and all motions sent by Monitor() are immediate motions (front of the queue). In this implementation all corrective motions are handled by Monitor() but this could be handled by another task or tasks. Note that these two tasks do not communicate directly but are synchronized through Alt_handler(). This structure of tasks represents a design philosophy in which one task sets the goals and another task or set of tasks independently contributes to achieving that goal. In a multi-sensor application, the monitoring tasks may be interpreting different sensors.

The description ignores initialization procedures and some of the "housekeeping" routines required, such as compensating for gravity loading on the IRCC in different orientations. Another task Panic monitors sensor data, looking for out-of-range conditions which could lead to damage of the sensor. Upon detection of any such state, the insertion process is terminated. The intent is to describe the main features of the insertion algorithm and demonstrate an approach to robot control with Harmony multiprocessor operating system.

The following elaboration of the MOTN_RQST struct gives for Alt_handler() the details of robot motion control.

Alt_handler(): Controlling Robot Motions from Harmony using ALTER

Alt_handler() provides a way for several client tasks to control the robot. It is set up as a server, so any task may open it as follows:

```
ucb = _Open( "ALTER:", 0 );
```

To create it use

```
id = _Server_create( ALTER, (struct INIT_REC *)&Alter_init );
```

Motion commands are specified in the MOTN_RQST struct. The following is a description of MOTN_RQST.

STD_REQUEST.MSG_TYPE:

UPDATE_RQST: What the client sends will not be placed on the queue. The reply contains the first element on the queue - the motion now in progress, as well as the setpoint data. This option alows the client to receive information from the handler without specifying any new action.

FLUSH_RQST: All elements on the motion queue are discarded. Any tasks that were waiting for a reply (signaling their motion is complete) are now permanently blocked, and should be destroyed.

UNBLOCK_RQST: The first element on the queue is removed, and the task that placed it there is replied to. This allows a motion to be terminated prematurely. Certain motions go indefinitely. In this case an unblock_rqst must be used to stop it.

MOTION_RQST: A command to be sent to Alt_server() is placed on the queue. One or more additional fields in the MOTN_RQST structure indicates the nature of the command. A description of these fields follows.

1. EXECUTE: Indicates where on the motion queue the request will be placed.

QUEUED: Motn_rqst is placed at the rear of the queue.
IMMEDIATE: Motn_rqst is placed at the front of the queue. It will be the next command sent to Alt_server().

2. MODE: This field selects one of five possible ALTER commands. They are:

IDLE: No motion. The position of the robot is unchanged. This calls Idle().
MOVE: The robot moves as specified in MOTN_RQST. This motion is done using Transrot().
START_ALTER: Starts the ALTER communications link. This should only be done once, before any other ALTER commands, or when following a STOP_ALTER. This calls Ini_Alter().

STOP_ALTER: Stops the robot and terminates communication with the controler. To resume communications, use START_ALTER. This calls Stp_Alter().

DEFINE_FIXED_FRAME: A frame found in FRAME_BUFF[] (passed by the user) is used to define a new fixed frame. All robot motions in FIXED frame are with respect to this newly defined coordinate system. As this option takes close to 28 ms to be executed, an Idle() is issued to the robot to satisfy the communications protocol.

3. FRAME: This field is needed whenever a MOVE is requested. It states the frame in which the motion is carried out.

WORLD: Motions are performed in the world coordinate system.

TOOL: Motions are performed in a dynamically changing tool coordinate system. The position of the tool, as given by the setpoint data received from the controller, is used to update this frame every 28 ms.

FIXED: Motions are performed with respect to a frame previously defined by a DEFINE_FIXED_FRAME command.

4. REPETITIONS: The motion is done REPETITIONS number of times. That is, REPETITIONS number of times Transrot() is called. The total distance traveled by the robot is REPETITIONS * MOTION_BUFF. The time taken to do this is REPETITIONS * 28ms. When REPETITIONS = -1 the motion is done indefinitely.

5. EXCEPTION: An exeption code which can be retreived by the VAL EXCEPTION function. This field is needed only when a MOVE is requested. It is usually 0.

6. FRAME_BUFF: This buffer contains a reference frame (specified by the user) used when mode = DEFINE_FIXED_FRAME. Only the first nine elements are needed. They are a rotation matrix scaled by 16384. These values must be int 16. The following is the order in which the matrix appears.

$$\begin{vmatrix} 0 & 3 & 6 \\ 1 & 4 & 7 \\ 2 & 5 & 8 \end{vmatrix}$$

7. MOTION_BUFF: This buffer contains the increments in each of the six directions. While a motion command is being executed, the robot moves by these increments every 28 ms.
They are floating point numbers. All translations are given in millimetres, and all rotations in degrees. This field is needed only when the mode is MOVE.

MOTION_BUFF[0]	translation in X.
MOTION_BUFF[1]	translation in Y.
MOTION_BUFF[2]	translation in Z.
MOTION_BUFF[3]	rotation about X.
MOTION_BUFF[4]	rotation about Y.
MOTION_BUFF[5]	rotation about Z.

Information is returned to the user program in the MOTN_RPLY struct. It is IMPORTANT to note that this information is one 28 ms cycle behind the actual state of the robot. The following is a description of this buffer.

MOTN_REQUEST: This is the first element on the motion queue. It is useful to know what motion is currently being executed. For example, MOTN_REQUEST.REPETITIONS indicates how far that motion is to completion.

ERROR: This field contains error codes or the state of ALTER returned by the ALTER server.

- ALTER_OK: Value returned when the communications is running smoothly. Also returned when ALTER is idling.
- ALTER_DOWN: If the ALTER link is stopped using STOP_ALTER, or the communications link was never initialized with START_ALTER, then this error code is returned at any attempted ALTER command.
- ALTER_UP: If the ALTER link has already been initialized, then any attempt to do this again (i.e., START_ALTER) will return this error code.

SETPOINT_BUFF: This field contains the transformation matrix returned by ALTER. It holds the current position and orientation of the tool tip. The first nine elements are a rotation matrix scaled by 16384. The last three elements are the displacements in world X, Y, and Z, scaled by 32.
This information can be used directly to define a fixed frame by copying the first nine elements into a

DEFINE_FIXED_FRAME MOTN_RQST. This matrix is updated every 28ms. SETPOINT_BUFF is in the following form:

$$\begin{vmatrix} 0 & 3 & 6 & 9 \\ 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \end{vmatrix}$$

SUMMARY

This report has outlined a procedure for the insertion of a rectangular shaped block into its tight fitting socket. For this demonstration, a PUMA 560 robot using the VAL II extension ALTER is controlled from a multiprocessor computer running the Harmony Operating system. Based on data from a tactile displacement sensor, total motion control is specified in the multiprocessor and setpoint Cartesian increments are passed to the PUMA controller. The program structure described here accomodates asynchronous and parallel operations and still insures efficient task synchronization as needed.

ACKNOWLEDGEMENT

The authors acknowledge the contribution of D. Green in hardware development and for software support related to the chorus machine utilization.

REFERENCES

1. W. Morven Gentleman. Message passing between sequential processes: the reply primitive and the administrator concept. Software-Practice and Experience, 11(5): 435-466; 1981.

2. D.A. Green. Chorus - a multiprocessor architecture for real-time control applications. NRC/ERB-964, National Research Council of Canada, Ottawa, Ont. December 1983.

3. W.M. Gentleman. Using the Harmony Operating System. NRC/ERB-966, National Research Council of Canada, Ottawa, Ont. December 1983. Revised October 1984, May 1985, March 1987.

4. D.H. O'Hara, S. Elgazzar, and G. The. ALTER-HARMONY: control of a PUMA robot from the Chorus multiprocessor. NRC/ERB-1002, National Research Council of Canada, Ottawa, Ont. November 1987.

5. The Model 4 instrumented remote center compliance. In CSDL Report No. R-1682 March 1984. C.S. Draper Laboratory Cambridge Mass. USA

6. R.P. Paul. Robot manipulators: mathematics, programming and control. The MIT Press 1981.

APPENDIX A

The code listed here has been edited. Its function is to aid in understanding routines only.

```
typedef struct STD_RQST
  {
    uint_16             MSG_SIZE;
    int_16              MSG_TYPE;
  };


typedef struct STD_RPLY
  {
    uint_16             MSG_SIZE;
    uint_32             RESULT;
  };


typedef struct OPEN_RPLY
  {
    struct STD_RPLY             STD_REPLY;   /* RESULT is an error code */
    uint_32                     UCB_SERVER;
    uint_32                     UCB_CONNECTION;
    uint_32                     UCB_FLAGS;
    uint_32                     SIZE_UCB_XTRA;
  };


typedef struct UCB                      /* user connection block */
  {
    struct UCB                  *NEXT_UCB;

    struct OPEN_RPLY            UCB_MAIN;
    char                        *UCB_XTRA;
  };


typedef struct DIR_RPLY
  {
    struct STD_RPLY     STD_REPLY;
    uint_32             DIR_SERVER;
  };
```

```
typedef struct CLOSE_RQST
  {
    struct STD_RQST     STANDARD;
    uint_32             CLOSE_CONNECTION;
  };


typedef struct OPEN_RQST
  {
    struct STD_RQST     STANDARD;
    uint_16             OPEN_USER_ID;
    char                OPEN_NAME[1];         /* space for name follows */
  };


typedef struct MOTN_RQST
  {
    struct STD_RQST             STD_REQUEST;
    int_16                      EXECUTE;
    int_16                      MODE;
    char                        FRAME;
    int_16                      REPETITIONS;
    char                        EXCEPTION;
    int_16                      FRAME_BUFF[12];
    float                       MOTION_BUFF[6];
  };


typedef struct ALT_RQST
  {
    struct STD_RQST             STD_REQUEST;
    int_16                      ERROR;
    int_16                      SETPOINT_BUFF[12];
  };


typedef struct MOTN_RPLY
  {
    struct MOTN_RQST            MOTN_REQUEST;
    int_16                      ERROR;
    int_16                      SETPOINT_BUFF[12];
  };
```

```
union HANDLER_MSG
  {
    struct STD_RQST             STD_REQUEST;
    struct MOTN_RQST            MOTN_REQUEST;
    struct ALT_RQST             ALT_REQUEST;
    struct OPEN_RQST            ALT_OPEN;
    struct CLOSE_RQST           ALT_CLOSE;
  };


typedef struct AIO_DATA
  {
    struct STD_RQST             STD_REQUEST;
    int_32                      DISPLACEMENTS[6];
    char                        *TEXT;
  };


struct DATA_RQST
  {
    struct STD_RQST             STD_REQUEST;
    int_16                      CONNECTION;
  };


struct TABLE
  {
    uint_32                     REQUESTOR;
    int_16                      VALID_DATA;
    int_16                      RPLY_PENDING;
  };


union AIO_MSG
  {
    struct STD_RQST             STD_REQUEST;
    struct DATA_RQST            DATA_REQUEST;
    struct AIO_DATA             FILTERED_DATA;
    struct OPEN_RQST            AIOH_OPEN;
    struct CLOSE_RQST           AIOH_CLOSE;
  };
```

```
struct GLOB_DATA
  {
    struct STD_RQST             STD_REQUEST;
    int_32                      OFFSETS[6];
    uint_32                     DISP_ID;
    uint_32                     PANIC_ID;
  };


struct TASK_TEMPLATE _Template_list[] =
  {
      { MAIN, main, 2048, 9, 0 },
      { DIRECTORY, _Directory, 436, 7, 0 },
      { GOSSIP, _Gossip, 648, 5, 0 },
      { TTY, _Tty_server, 584, 5, 0 },
      { TTI, _Tti, 282, 0, 0 },
      { TTO, _Tto, 312, 0, 0 },
      { CLOCK, _Clock_server, 500, 6, 0 },
      { CLOCK_NOTIFIER, _Clock_notifier, 370, 1, 0 },
      { SEQUENCE, Sequence, 2000, 7, 0 },
      { MONITOR, Monitor, 2000, 8, 0 },
      { 0, 0, 0, 0, 0 }
  };


struct INT_PAIR _Interrupt_list[] =
  {
    { 4, _Ptm_int },
    { 5, _Serial_int },
    { 0, 0 }
  };
```

```
*                          BLOCK INSERTION DEMO

struct UCB *ucb_it = 0, *ucb_ot = 0;
extern struct UCB *ucb_iv, *ucb_ov;
struct UCB *ucb_aio = 0;

int_32 offsets[3][6] = 0;
int_32 curr_offsets[6] = 0;
float x = 0, y = 0, z = 0, rx = 0, ry = 0, rz = 0;

main()
{
   extern struct TTYI_PORT_INIT_REC P0_tty0_init;
   extern struct EHVT_PORT_INIT_REC Ehvt0_init;
   extern struct AIO_INIT_REC Aio_init;
   extern struct CLK_INIT_REC Clock_init;
   extern struct ALTER_INIT_REC Alter_init;
   extern struct AIOH_INIT_REC Aioh_init;

   struct STD_RQST request;
   struct STD_RPLY reply;
   struct AIO_DATA aio_data;
   struct MOTN_RPLY motn_rply;
   struct MOTN_RQST motn_rqst;
   struct DATA_RQST data_rqst;
   struct GLOB_DATA glob_data;
   struct UCB *alter_ucb, *aioh_ucb;
   uint_32 id, mon_id, seq_id, panic_id, disp_id, disp_cour_id;
   char c, cd;
   int j, i = 1;
   char message[180];



   id = _Server_create( TTY, (struct INIT_REC *)&P0_tty0_init );
   id = _Server_create( EHVT, (struct INIT_REC *)&Ehvt0_init );
   id = _Server_create( AIO, (struct INIT_REC *)&Aio_init );
   id = _Server_create( CLOCK, (struct INIT_REC *)&Clock_init );
   id = _Server_create( ALTER, (struct INIT_REC *)&Alter_init );
   id = _Server_create( AIOHAND, (struct INIT_REC *)&Aioh_init );

   _Selectinput( ucb_it = _Open( "P0_TTY0: +r", 0 ) );

   _Selectoutput( ucb_ot = _Open( "P0_TTY0: +w", 0 ) );
   ucb_iv = _Open( "EHVT: +r", 0 );
```

```
ucb_ov = _Open( "EHVT: +w", 0 );
ucb_aio = _Open( "AIO: +rw", 0 );

_Putstr( "\n\nTTY, AIO, and EHVT now opened.\n\n");
_Flush();
while( 1 && ( ( j = delay( DT,message ) ) != 62 ) )
{
     _Printf( " delay reply = %i \n Continue ? (Y/N)  ", j );
     _Flush();
     if ( ( c = _Get() ) != '\n' ) cd = _Get();
     _Putstr( "\n" );
     _Flush();
     if ( c == 'Y' ) i = 0;
}
_Breakpoint( "starting execution" );


 disp_id = _Create( DISPLAY );

 glob_data.STD_REQUEST.MSG_SIZE = sizeof( struct GLOB_DATA );
 glob_data.DISP_ID = disp_id;
 reply.MSG_SIZE = sizeof( struct STD_RPLY );

 alter_ucb = _Open( "ALTER:", 0 );
 aioh_ucb = _Open( "AIOHAND:", 0 );

 move( START, message );
 Start_alter( alter_ucb );


 Get_offsets( aioh_ucb, alter_ucb );

 for( i=0; i < CHANNELS; i++ )
   glob_data.OFFSETS[i] = curr_offsets[i];

 panic_id = _Create( PANIC );
 glob_data.PANIC_ID = panic_id;
 _Send( (char *)&glob_data, (char *)&reply, panic_id );

 disp_cour_id = _Create( DISCOUR1 );
 _Send( (char *)&glob_data, (char *)&reply, disp_cour_id );

 seq_id = _Create( SEQUENCE );
 _Send( (char *)&glob_data, (char *)&reply, seq_id );
```

```
   mon_id = _Create( MONITOR );
   _Send( (char *)&glob_data, (char *)&reply, mon_id );


   _Receive( (char *)&request, mon_id );

   Stop_alter( alter_ucb );

   _Receive( (char *)&request, disp_id );
};


Start_alter ( ucb )

struct UCB *ucb;
{

   char message[180];
   int_16 i;
   struct MOTN_RQST mot_rqst;
   struct MOTN_RPLY mot_rply;

   mot_rqst.STD_REQUEST.MSG_TYPE = MOTION_RQST;
   mot_rqst.EXECUTE = IMMEDIATE;
   mot_rqst.MODE = START_ALTER;
   mot_rqst.STD_REQUEST.MSG_SIZE = sizeof( mot_rqst );
   mot_rply.MOTN_REQUEST.STD_REQUEST.MSG_SIZE = sizeof( mot_rply );
   _Send( (char *)&mot_rqst, (char *)&mot_rply,
          ucb->UCB_MAIN.UCB_SERVER );
                                                  /* Synchronize with */
                                                  /* Ini_Alter.       */
   pcabort( message );
   execute( "demoalt", message );
};
```

MONITOR TASK:

```
/*
 *  This task reads back values from the sensor. It uses these to detect
 *  the end of each motion sequence, and also remove any bindings that may
 *  occur.
 *  The gross motion steps are found in the task Sequence().
 *  Synchronization between Monitor() and Sequence() is done through
 *  Alt_handler(), the robot motion server, using the unblock command.
 *  All motions sent from Monitor() are in IMMEDIATE mode to remove
 *  possible bindings.
 */

Monitor()
{
  struct STD_RQST std_rqst;
  struct STD_RPLY std_rply;
  struct AIO_DATA aio_data;
  struct GLOB_DATA glob_data;
  struct UCB *aioh_ucb, *alt_ucb;
  struct MOTN_RQST unblock, update, wrldz, wrldx, tooly,
                   wrldnx, toolrz, toolxyz, toolnx;
 struct MOTN_RPLY motn_rply, in_socket;
 int_16 noend;
 uint_32 alter_id;

/*  Get id of the display task from father.  */

  glob_data.STD_REQUEST.MSG_SIZE = sizeof( struct GLOB_DATA );
  std_rply.MSG_SIZE = sizeof( struct STD_RPLY );
  _Receive( (char *)&glob_data, _Father_id() );
  _Reply( (char *)&std_rply, _Father_id() );

/*  Initialize motion structures used  */

  I_wrldx( &wrldx );
  wrldx.REPETITIONS = 2;
  wrldx.EXECUTE = IMMEDIATE;
  wrldx.MOTION_BUFF[0] = .1;

  I_wrldz( &wrldz );
  wrldz.REPETITIONS = 1;
  wrldz.EXECUTE = IMMEDIATE;
```

```
I_wrldnx( &wrldnx );
wrldnx.REPETITIONS = 1;
wrldnx.EXECUTE = IMMEDIATE;

I_toolrz( &toolrz );
toolrz.REPETITIONS = 1;
toolrz.EXECUTE = IMMEDIATE;

I_toolnx( &toolnx );
toolnx.REPETITIONS = 1;
toolnx.EXECUTE = IMMEDIATE;

I_tooly( &toolxyz );
toolxyz.REPETITIONS = 2;
toolxyz.EXECUTE = IMMEDIATE;
toolxyz.MOTION_BUFF[0] = -.05;

toolxyz.MOTION_BUFF[1] = .05;
toolxyz.MOTION_BUFF[2] = -.05;

I_tooly( &tooly );
tooly.REPETITIONS = 1;
tooly.EXECUTE = IMMEDIATE;
tooly.MOTION_BUFF[1] = .1;

unblock.STD_REQUEST.MSG_SIZE = sizeof( struct MOTN_RQST );
unblock.STD_REQUEST.MSG_TYPE = UNBLOCK_RQST;

 update.STD_REQUEST.MSG_SIZE = sizeof( struct MOTN_RQST );
 update.STD_REQUEST.MSG_TYPE = UPDATE_RQST;

 motn_rply.MOTN_REQUEST.STD_REQUEST.MSG_SIZE =
      sizeof( struct MOTN_RPLY );
 in_socket.MOTN_REQUEST.STD_REQUEST.MSG SIZE =
       sizeof( struct MOTN_RPLY );

/*  Open connection to aio handler and alt handler */

 aioh_ucb = _Open( "AIOHAND:", 0 );
 alt_ucb = _Open( "ALTER:", 0 );
 alter_id = alt_ucb->UCB_MAIN.UCB_SERVER;
```

```
/******************************** Step 1 ***********************************/

   do
   {
     Get_displac( &aio_data, aioh_ucb );
   }
   while( z > -.15 );


   _Send( (char *)&unblock, (char *)&motn_rply, alter_id );
   To_screen( glob_data.DISP_ID, &aio_data,
              "STEP1: point contact with flange\n" );
   _Send( (char *)&unblock, (char *)&motn_rply, alter_id );

/******************************** Step 2 ***********************************/

   do
   {
     Get_displac( &aio_data, aioh_ucb );
   }
   while( z < -.1 );

   _Send( (char *)&unblock, (char *)&in_socket, alter_id );
   To_screen( glob_data.DISP_ID, &aio_data,
              "STEP2: point falls in socket\n" )
     Send( (char *)&unblock, (char *)&in socket, alter_id );

/******************************** Step 3 ***********************************/

   do
   {
     noend = FALSE;
     Get_displac( &aio_data, aioh_ucb );

     if( rz > .2 || x > 4 )
     {
       noend = TRUE;
       _Send( (char *)&wrldx, (char *)&motn_rply, alter_id );
     }
   }
   while( z > -.3 || noend );

    Send( (char *)&unblock, (char *)&motn_rply, alter_id );

   do
   {
```

```
      Send( (char *)&toolxyz, (char *)&motn_rply, alter_id );
     Get_displac( &aio_data, aioh_ucb );
   }
   while( y > 4 || z < -.2 );

  To_screen( glob_data.DISP_ID, &aio_data, "STEP3: contact with back\n" );
  _Send( (char *)&unblock, (char *)&motn_rply, alter_id );

/******************************** Step 4 ***********************************/

  do
  {
    Get_displac( &aio_data, aioh_ucb );
  }
  while( x > -5 );

  _Send( (char *)&unblock, (char *)&motn_rply, alter_id );

  do
  {
    Get_displac( &aio_data, aioh_ucb );
  }
  while( x < -2 );

  _Send( (char *)&unblock, (char *)&motn_rply, alter_id );
  To_screen( glob_data.DISP_ID, &aio_data, "STEP4: edge contact\n" );
  _Send( (char *)&unblock, (char *)&motn_rply, alter_id );

/******************************** Step 5 ***********************************/

  do
  {
    Get_displac( &aio_data, aioh_ucb );
    while( y < 4 || y > 8 )
    {
      wrldnx.MOTION_BUFF[0] = (y-6) / 50;
      _Send( (char *)&wrldnx, (char *)&motn_rply, alter_id );
      Get_displac( &aio_data, aioh_ucb );
    }
  }
  while( x > -4 );

   Send( (char *)&unblock, (char *)&motn_rply, alter_id );

  do
```

```
  {
    _Send( (char *)&toolnx, (char *)&motn_rply, alter_id );
    Get_displac( &aio_data, aioh_ucb );
  }
  while( x < -5 );

  To_screen( glob_data.DISP_ID, &aio_data, "STEP5: corner contact\n" );
  _Send( (char *)&unblock, (char *)&motn_rply, alter_id );

/****************************** Step 6 **********************************/

  do
  {
    noend = FALSE;
    _Send( (char *)&update, (char *)&motn_rply, alter_id );
    Get_displac( &aio_data, aioh_ucb );
    if( (y > 4) &&
        (in_socket.SETPOINT_BUFF[11] - motn_rply.SETPOINT_BUFF[11] > 32) )
    {
      _Send( (char *)&tooly, (char *)&motn_rply, alter_id );
      noend = TRUE;
    }
    else if( (x < -5) &&
         (in_socket.SETPOINT_BUFF[11] - motn_rply.SETPOINT_BUFF[11] > 32) )
    {
      _Send( (char *)&toolnx, (char *)&motn_rply, alter_id );
      noend = TRUE;
    }
  }
  while( in_socket.SETPOINT_BUFF[11] - motn_rply.SETPOINT_BUFF[11] > 32
         || noend );

  _Send( (char *)&unblock, (char *)&motn_rply, alter_id );
  To_screen( glob_data.DISP_ID, &aio_data, "STEP6: above corner\n" );
  _Send( (char *)&unblock, (char *)&motn_rply, alter_id );

/****************************** Step 7 **********************************/

  do
  {
     Send( (char *)&update, (char *)&motn_rply, alter_id );
  }
  while( motn_rply.MOTN_REQUEST.REPETITIONS > 0 );

  To_screen( glob_data.DISP_ID, &aio_data, "STEP7: above socket\n" );
   Send( (char *)&unblock, (char *)&motn_rply, alter_id );
```

```
/******************************** Step 8 ***********************************/

  _Destroy( glob_data.PANIC_ID );

  do
  {
    Get_displac( &aio_data, aioh_ucb );

  }
  while( z > -.3 );

  _Send( (char *)&unblock, (char *)&motn_rply, alter_id );

  do
  {
    Get_displac( &aio_data, aioh_ucb );
  }
  while( z < -.1 );

  _Send( (char *)&unblock, (char *)&motn_rply, alter_id );
  To_screen( glob_data.DISP_ID, &aio_data, "STEP8: in socket\n" );

/*****************************************************************************/

/*  Block now in socket --- send to main  */

  _Send( (char *)&std_rqst, (char *)&std_rply, _Father_id() );


};
```

SEQUENCE TASK:

```
/*
 *  This task contains the gross motions needed to insert the block.
 *  Each individual motion is of the form move straight in a specific
    direction forever.  Another task ( Monitor ) watches the sensor to
    determine the transition between motion sequences, and unblocks this
    task via Alt_handler(), the robot motion server.
 *  Each motion is followed by an idle for synchronization purposes.
 *  Offsets are changed by this task.
 */

Sequence()
{
  extern double arctan(), sqrt();

  struct MOTN_RQST motn_rqst;
  struct MOTN_RPLY motn_rply;
  struct GLOB_DATA glob_data;
  struct STD_RPLY std_rply;
  struct UCB *alter_ucb;
  double theta, psi, sin_theta, cos_theta, sin_psi, cos_psi;
  double deg = 57.2957795;
  uint_32 alter_id;

/*************************************************************************/

/*  Get id of the display task from father  */

  glob_data.STD_REQUEST.MSG_SIZE = sizeof( struct GLOB_DATA );
  std_rply.MSG_SIZE = sizeof( struct STD_RPLY );
  _Receive( (char *)&glob_data, _Father_id() );
  _Reply( (char *)&std_rply, _Father_id() );

/*  open connection to alter server  */

  alter_ucb = _Open( "ALTER:", 0 );
  alter_id = alter_ucb->UCB_MAIN.UCB_SERVER;

/******************************** Step 1 *********************************/

/*
 *  In this step the arm is decelerated down a fixed distance, and then *
 the motion continues down indefinitely. ( that is until the Monitor *
 task detects an end condition )
 */
```

```
  Change_offsets( 1, &glob_data );

  motn_rqst.STD_REQUEST.MSG_SIZE = sizeof( struct MOTN_RQST );
  motn_rqst.STD_REQUEST.MSG_TYPE = MOTION_RQST;
  motn_rqst.EXECUTE = QUEUED;
  motn_rqst.MODE = MOVE;
  motn_rqst.FRAME = WORLD;
  motn_rqst.REPETITIONS = 1;
  motn_rqst.EXCEPTION = 0;
  motn_rqst.MOTION_BUFF[0] = 0;
  motn_rqst.MOTION_BUFF[1] = 0;
  motn_rqst.MOTION_BUFF[2] = -2;
  motn_rqst.MOTION_BUFF[3] = 0;
  motn_rqst.MOTION_BUFF[4] = 0;
  motn_rqst.MOTION_BUFF[5] = 0;


  motn_rply.MOTN_REQUEST.STD_REQUEST.MSG_SIZE = sizeof( struct MOTN_RPLY );

  while( motn_rqst.MOTION_BUFF[2] < -.05 )
    {
      _Send( (char *)&motn_rqst, (char *)&motn_rply, alter_id );
      motn_rqst.MOTION_BUFF[2] += .01;
    }

  I_wrldnz( &motn_rqst );
  _Send( (char *)&motn_rqst, (char *)&motn_rply, alter_id );

  I_idle( &motn_rqst );
  _Send( (char *)&motn_rqst, (char *)&motn_rply, alter_id );

/********************************* Step 2 *********************************/

  I_wrldx( &motn_rqst );
   Send( (char *)&motn_rqst, (char *)&motn_rply, alter_id );

  I_idle( &motn_rqst );
   Send( (char *)&motn_rqst, (char *)&motn_rply, alter_id );

/********************************* Step 3 *********************************/

  I_toolz( &motn_rqst );
   Send( (char *)&motn_rqst, (char *)&motn_rply, alter_id );

  I_idle( &motn_rqst );
   Send( (char *)&motn_rqst, (char *)&motn_rply, alter_id );
```

```
/********************************* Step 4 *********************************/

  I_toolrz( &motn_rqst );
  _Send( (char *)&motn_rqst, (char *)&motn_rply, alter_id );

  I_toolrnz( &motn_rqst );
  _Send( (char *)&motn_rqst, (char *)&motn_rply, alter_id );

  I_idle( &motn_rqst );
  _Send( (char *)&motn_rqst, (char *)&motn_rply, alter_id );

/********************************* Step 5 *********************************/

  I_wrldy( &motn_rqst );
  _Send( (char *)&motn_rqst, (char *)&motn_rply, alter_id );

  I_idle( &motn_rqst );
  _Send( (char *)&motn_rqst, (char *)&motn_rply, alter_id );

/********************************* Step 6 *********************************/

  I_toolnz( &motn_rqst );
   Send( (char *)&motn_rqst, (char *)&motn_rply, alter_id );

  I_idle( &motn_rqst );
   Send( (char *)&motn_rqst, (char *)&motn_rply, alter_id );

/********************************* Step 7 *********************************/

/*
 *  This motion is different from the rest. It is a precomputed move for a
 *  given rotation. No sensor data is needed to detect the end of this
 *  motion sequence.
 */

  sin_theta = (double)motn_rply.SETPOINT_BUFF[6] / -16384;
  cos_theta = sqrt( 1 - sin_theta * sin_theta );
  theta = deg * arctan( sin_theta / cos_theta );

  sin_psi = (double)motn_rply.SETPOINT_BUFF[7] / cos_theta / 16384;
  cos_psi = sqrt( 1 - sin_psi * sin_psi );
  psi = deg * arctan( sin_psi / cos_psi );

  motn_rqst.MODE = MOVE;
  motn_rqst.FRAME = WORLD;
```

```
  motn_rqst.REPETITIONS = 100;
  motn_rqst.EXCEPTION = 0;
  motn_rqst.MOTION_BUFF[0] = 0;
  motn_rqst.MOTION_BUFF[1] = 0;
  motn_rqst.MOTION_BUFF[2] = 0;
  motn_rqst.MOTION_BUFF[3] = -psi / 100;
  motn_rqst.MOTION_BUFF[4] = -theta / 100;
  motn_rqst.MOTION_BUFF[5] = 0;

  _Send( (char *)&motn_rqst, (char *)&motn_rply, alter_id );

  I_idle( &motn_rqst );
  _Send( (char *)&motn_rqst, (char *)&motn_rply, alter_id );

/********************************* Step 8 *********************************/

  Change_offsets( 0, &glob_data );

  I_wrldnz( &motn_rqst );
  motn_rqst.MOTION_BUFF[2] = -.1;
   Send( (char *)&motn_rqst, (char *)&motn_rply, alter_id );

  I_wrldz( &motn_rqst );
   Send( (char *)&motn_rqst, (char *)&motn_rply, alter_id );

  I_idle( &motn_rqst );
   Send( (char *)&motn_rqst, (char *)&motn_rply, alter_id );

/*****************************************************************************/

};


I_idle( mot_rqst )

struct MOTN_RQST *mot_rqst;
{

  mot_rqst->STD_REQUEST.MSG_TYPE = MOTION_RQST;
  mot_rqst->EXECUTE = QUEUED;
  mot_rqst->MODE = IDLE;
  mot_rqst->REPETITIONS = -1;
  mot_rqst->EXCEPTION = 0;
  mot_rqst->STD_REQUEST.MSG_SIZE = sizeof( struct MOTN_RQST );

};
```

```
I_wrldx( mot_rqst )

struct MOTN_RQST *mot_rqst;
{

  mot_rqst->STD_REQUEST.MSG_TYPE = MOTION_RQST;
  mot_rqst->EXECUTE = QUEUED;
  mot_rqst->MODE = MOVE;
  mot_rqst->FRAME = WORLD;
  mot_rqst->REPETITIONS = -1;
  mot_rqst->EXCEPTION = 0;
  mot_rqst->MOTION_BUFF[0] = .05;
  mot_rqst->MOTION_BUFF[1] = 0;
  mot_rqst->MOTION_BUFF[2] = 0;
  mot_rqst->MOTION_BUFF[3] = 0;
  mot_rqst->MOTION_BUFF[4] = 0;
  mot_rqst->MOTION_BUFF[5] = 0;
  mot_rqst->STD_REQUEST.MSG_SIZE = sizeof( struct MOTN_RQST );

};


I_wrldnx( mot_rqst )

struct MOTN_RQST *mot_rqst;
{

  mot_rqst->STD_REQUEST.MSG_TYPE = MOTION_RQST;
  mot_rqst->EXECUTE = QUEUED;
  mot_rqst->MODE = MOVE;
  mot_rqst->FRAME = WORLD;
  mot_rqst->REPETITIONS = -1;
  mot_rqst->EXCEPTION = 0;
  mot_rqst->MOTION_BUFF[0] = -.05;
  mot_rqst->MOTION_BUFF[1] = 0;
  mot_rqst->MOTION_BUFF[2] = 0;
  mot_rqst->MOTION_BUFF[3] = 0;
  mot_rqst->MOTION_BUFF[4] = 0;
  mot_rqst->MOTION_BUFF[5] = 0;
  mot_rqst->STD_REQUEST.MSG_SIZE = sizeof( struct MOTN_RQST );

};
```

```
I_wrldy( mot_rqst )

struct MOTN_RQST *mot_rqst;
{

  mot_rqst->STD_REQUEST.MSG_TYPE = MOTION_RQST;
  mot_rqst->EXECUTE = QUEUED;
  mot_rqst->MODE = MOVE;
  mot_rqst->FRAME = WORLD;
  mot_rqst->REPETITIONS = -1;
  mot_rqst->EXCEPTION = 0;
  mot_rqst->MOTION_BUFF[0] = 0;
  mot_rqst->MOTION_BUFF[1] = .15;
  mot_rqst->MOTION_BUFF[2] = 0;
  mot_rqst->MOTION_BUFF[3] = 0;
  mot_rqst->MOTION_BUFF[4] = 0;
  mot_rqst->MOTION_BUFF[5] = 0;
  mot_rqst->STD_REQUEST.MSG_SIZE = sizeof( struct MOTN_RQST );

};


I_wrldz( mot_rqst )

struct MOTN_RQST *mot_rqst;
{

  mot_rqst->STD_REQUEST.MSG_TYPE = MOTION_RQST;
  mot_rqst->EXECUTE = QUEUED;
  mot_rqst->MODE = MOVE;
  mot_rqst->FRAME = WORLD;
  mot_rqst->REPETITIONS = -1;
  mot_rqst->EXCEPTION = 0;
  mot_rqst->MOTION_BUFF[0] = 0;
  mot_rqst->MOTION_BUFF[1] = 0;
  mot_rqst->MOTION_BUFF[2] = .05;
  mot_rqst->MOTION_BUFF[3] = 0;
  mot_rqst->MOTION_BUFF[4] = 0;
  mot_rqst->MOTION_BUFF[5] = 0;
  mot_rqst->STD_REQUEST.MSG_SIZE = sizeof( struct MOTN_RQST );

};
```

```
I_wrldnz( mot_rqst )

struct MOTN_RQST *mot_rqst;
{

  mot_rqst->STD_REQUEST.MSG_TYPE = MOTION_RQST;
  mot_rqst->EXECUTE = QUEUED;
  mot_rqst->MODE = MOVE;
  mot_rqst->FRAME = WORLD;
  mot_rqst->REPETITIONS = -1;
  mot_rqst->EXCEPTION = 0;
  mot_rqst->MOTION_BUFF[0] = 0;
  mot_rqst->MOTION_BUFF[1] = 0;
  mot_rqst->MOTION_BUFF[2] = -.05;
  mot_rqst->MOTION_BUFF[3] = 0;
  mot_rqst->MOTION_BUFF[4] = 0;
  mot_rqst->MOTION_BUFF[5] = 0;
  mot_rqst->STD_REQUEST.MSG_SIZE = sizeof( struct MOTN_RQST );

};


I_toolnx( mot_rqst )

struct MOTN_RQST *mot_rqst;
{

  mot_rqst->STD_REQUEST.MSG_TYPE = MOTION_RQST;
  mot_rqst->EXECUTE = QUEUED;
  mot_rqst->MODE = MOVE;
  mot_rqst->FRAME = TOOL;
  mot_rqst->REPETITIONS = -1;
  mot_rqst->EXCEPTION = 0;
  mot_rqst->MOTION_BUFF[0] = -.05;
  mot_rqst->MOTION_BUFF[1] = 0;
  mot_rqst->MOTION_BUFF[2] = 0;
  mot_rqst->MOTION_BUFF[3] = 0;
  mot_rqst->MOTION_BUFF[4] = 0;
  mot_rqst->MOTION_BUFF[5] = 0;
  mot_rqst->STD_REQUEST.MSG_SIZE = sizeof( struct MOTN_RQST );

};
```

```
I_tooly( mot_rqst )

struct MOTN_RQST *mot_rqst;
{

  mot_rqst->STD_REQUEST.MSG_TYPE = MOTION_RQST;
  mot_rqst->EXECUTE = QUEUED;
  mot_rqst->MODE = MOVE;
  mot_rqst->FRAME = TOOL;
  mot_rqst->REPETITIONS = -1;
  mot_rqst->EXCEPTION = 0;
  mot_rqst->MOTION_BUFF[0] = 0;
  mot_rqst->MOTION_BUFF[1] = .05;
  mot_rqst->MOTION_BUFF[2] = 0;
  mot_rqst->MOTION_BUFF[3] = 0;
  mot_rqst->MOTION_BUFF[4] = 0;
  mot_rqst->MOTION_BUFF[5] = 0;
  mot_rqst->STD_REQUEST.MSG_SIZE = sizeof( struct MOTN_RQST );

};


I_toolz( mot_rqst )

struct MOTN_RQST *mot_rqst;
{

  mot_rqst->STD_REQUEST.MSG_TYPE = MOTION_RQST;
  mot_rqst->EXECUTE = QUEUED;
  mot_rqst->MODE = MOVE;
  mot_rqst->FRAME = TOOL;
  mot_rqst->REPETITIONS = -1;
  mot_rqst->EXCEPTION = 0;
  mot_rqst->MOTION_BUFF[0] = 0;
  mot_rqst->MOTION_BUFF[1] = 0;
  mot_rqst->MOTION_BUFF[2] = .05;
  mot_rqst->MOTION_BUFF[3] = 0;
  mot_rqst->MOTION_BUFF[4] = 0;
  mot_rqst->MOTION_BUFF[5] = 0;
  mot_rqst->STD_REQUEST.MSG_SIZE = sizeof( struct MOTN_RQST );

};
```

```
I_toolnz( mot_rqst )

struct MOTN_RQST *mot_rqst;
{

  mot_rqst->STD_REQUEST.MSG_TYPE = MOTION_RQST;
  mot_rqst->EXECUTE = QUEUED;
  mot_rqst->MODE = MOVE;
  mot_rqst->FRAME = TOOL;
  mot_rqst->REPETITIONS = -1;
  mot_rqst->EXCEPTION = 0;
  mot_rqst->MOTION_BUFF[0] = 0;
  mot_rqst->MOTION_BUFF[1] = 0;
  mot_rqst->MOTION_BUFF[2] = -.1;
  mot_rqst->MOTION_BUFF[3] = 0;
  mot_rqst->MOTION_BUFF[4] = 0;
  mot_rqst->MOTION_BUFF[5] = 0;
  mot_rqst->STD_REQUEST.MSG_SIZE = sizeof( struct MOTN_RQST );

};


I_toolrz( mot_rqst )

struct MOTN_RQST *mot_rqst;
{

  mot_rqst->STD_REQUEST.MSG_TYPE = MOTION_RQST;
  mot_rqst->EXECUTE = QUEUED;
  mot_rqst->MODE = MOVE;
  mot_rqst->FRAME = TOOL;
  mot_rqst->REPETITIONS = -1;
  mot_rqst->EXCEPTION = 0;
  mot_rqst->MOTION_BUFF[0] = 0;
  mot_rqst->MOTION_BUFF[1] = 0;
  mot_rqst->MOTION_BUFF[2] = 0;
  mot_rqst->MOTION_BUFF[3] = 0;
  mot_rqst->MOTION_BUFF[4] = 0;
  mot_rqst->MOTION_BUFF[5] = .05;
  mot_rqst->STD_REQUEST.MSG_SIZE = sizeof( struct MOTN_RQST );

};
```

```
I_toolrnz( mot_rqst )

struct MOTN_RQST *mot_rqst;
{

  mot_rqst->STD_REQUEST.MSG_TYPE = MOTION_RQST;
  mot_rqst->EXECUTE = QUEUED;
  mot_rqst->MODE = MOVE;
  mot_rqst->FRAME = TOOL;
  mot_rqst->REPETITIONS = -1;
  mot_rqst->EXCEPTION = 0;
  mot_rqst->MOTION_BUFF[0] = 0;
  mot_rqst->MOTION_BUFF[1] = 0;
  mot_rqst->MOTION_BUFF[2] = 0;
  mot_rqst->MOTION_BUFF[3] = 0;
  mot_rqst->MOTION_BUFF[4] = 0;
  mot_rqst->MOTION_BUFF[5] = -.05;
  mot_rqst->STD_REQUEST.MSG_SIZE = sizeof( struct MOTN_RQST );

};


To_screen ( disp_id, aio_data, text )

uint_32 disp_id;
struct AIO_DATA *aio_data;
char *text;
{
  struct STD_RPLY std_rply;

  std_rply.MSG_SIZE = sizeof( std_rply );
  aio_data->STD_REQUEST.MSG_SIZE = sizeof( struct AIO_DATA );
  aio_data->STD_REQUEST.MSG_TYPE = PRINT_VALUES;
  aio_data->TEXT = text;

  _Send( (char *)aio_data, (char *)&std_rply, disp_id );
};


typedef struct STD_RQST
  {
    uint_16             MSG_SIZE;
    int_16              MSG_TYPE;
  };
```

```
typedef struct STD_RPLY
  {
    uint_16             MSG_SIZE;
    uint_32             RESULT;
  };


typedef struct OPEN_RPLY
  {
    struct STD_RPLY             STD_REPLY;    /* RESULT is an error code */
    uint_32                     UCB_SERVER;
    uint_32                     UCB_CONNECTION;
    uint_32                     UCB_FLAGS;
    uint_32                     SIZE_UCB_XTRA;
  };


typedef struct CLOSE_RQST
  {
    struct STD_RQST     STANDARD;
    uint_32             CLOSE_CONNECTION;
  };


typedef struct OPEN_RQST
  {
    struct STD_RQST     STANDARD;
    uint_16             OPEN_USER_ID;
    char                OPEN_NAME[1];          /* space for name follows */
  };


/*
 *  Alter handler initialization structure.
 */

struct ALTER_INIT_REC
  {
    struct INIT_REC     ALTI_HDR;
    char                ALTI_NAME[32];
  };
```

```
typedef struct MOTN_RQST
  {
    struct STD_RQST          STD_REQUEST;
    int_16                   EXECUTE;
    int_16                   MODE;
    char                     FRAME;
    int_16                   REPETITIONS;
    char                     EXCEPTION;
    int_16                   FRAME_BUFF[12];
    float                    MOTION_BUFF[6];
  };


typedef struct ALT_RQST
  {
    struct STD_RQST          STD_REQUEST;
    int_16                   ERROR;
    int_16                   SETPOINT_BUFF[12];
  };


typedef struct MOTN_RPLY
  {
    struct MOTN_RQST         MOTN_REQUEST;
    int_16                   ERROR;
    int_16                   SETPOINT_BUFF[12];
  };


struct QCELL
  {
    struct MOTN_RQST         MOTN_REQUEST;
    uint_32                  REQUESTOR;
    struct QCELL             *NEXT;
  };


struct QUEUE
  {
    struct QCELL             *FRONT;
    struct QCELL             *REAR;
  };
```

```
union HANDLER_MSG
  {
    struct STD_RQST              STD_REQUEST;
    struct MOTN_RQST             MOTN_REQUEST;
    struct ALT_RQST              ALT_REQUEST;
    struct OPEN_RQST             ALT_OPEN;
    struct CLOSE_RQST            ALT_CLOSE;
  };


struct TASK_TEMPLATE _Template_list[] =
  {
      { EHVT, _Ehvt_server, 1100, 1, 0 },
      { EHVTIP, _Ehvtip, 400, 0, 0 },
      { EHVTIC, _Ehvtic, 400, 2, 0 },
      { EHVTO, _Tto, 132, 0, 0 },
      { _OUT_PACKET, _Out_packet, 400, 4, 0 },
      { _IN_PACKET, _In_packet, 400, 3, 0 },
      { ALTSERV, Alt_server, 700, 5, 0 },
      { ALTCOUR, Alt_courier, 700, 6, 0 },
      { ALTER, Alt_handler, 1000, 7, 0 },
      { PANIC, Panic_stop, 1000, 8, 0 },
      { 0, 0, 0, 0, 0 }
  };


struct INT_PAIR _Interrupt_list[] =
  {
    { 4, _Ptm_int },
    { 5, _Serial_int },
    { 7, Vavpint },
    { 0, 0 }
  };
```

```
* Alt_handler


/*
 *  This task receives motions from clients and sends them to Alt_server
 *  via Alt_courier.
 *  Clients open a stream to Alt_handler and then sends motion requests.
 *  Only when the request is completed, such as a motion finished,
 *  does Alt_handler reply and unblock the client.
 *  Motion requests are placed on a queue, either at the front, or the
 *  rear, depending wheather the immediate or queued option is selected.
 *  Replies to a client contain the current motion being executed,
 *  as well as setpoint data returned by alter from the val controler.
 *  Alt_courier sends requests to this task ( ALTER_RQST ) every 28ms.
 *  The reply to Alt_server contains the first element on the motion queue.
 */

Alt_handler()
{
  struct QUEUE freecells, motions;
  struct QCELL *cell;
  struct MOTN_RQST idle_rqst;
  struct MOTN_RPLY motn_rply;
  struct STD_RQST std_rqst;
  struct STD_RPLY std_rply;
  struct ALTER_INIT_REC alter_init;
  struct OPEN_RPLY open_rply;
  union HANDLER_MSG request;
  uint_32 id, requestor;

/*
 *  Initialize the Alter handler.
 */

std_rqst.MSG_SIZE = sizeof( struct STD_RQST );
std_rqst.MSG_TYPE = INITIALIZE_SERVER;
alter_init.ALTI_HDR.STANDARD.MSG_SIZE = sizeof( struct ALTER_INIT_REC );
_Send( (char *)&std_rqst, (char *)&alter_init, _Father_id() );
if ( alter_init.ALTI_HDR.STANDARD.MSG_TYPE != ALTIT_ALTER_INIT_REC )
    _Abort( "Alt_handler: bad init type. \n" );
```

```
Report_for_service( alter_init.ALTI_NAME, REPORT_FOR_SERVICE );

 id = _Create( ALTCOUR );

 Init_q( &freecells );
 Init_q( &motions );

 idle_rqst.MODE = IDLE;
 idle_rqst.EXCEPTION = 0;

 for(;;)
 {
   request.STD_REQUEST.MSG_SIZE = sizeof( union HANDLER_MSG );
   requestor = _Receive( (char *)&request, 0 );

   switch( request.STD_REQUEST.MSG_TYPE )
   {


     case ALTER_RQST:
       _Copy( &request.ALT_REQUEST.ERROR, &motn_rply.ERROR,
              13 * sizeof( int_16 ) );

       if ( !Empty( &motions ) )
       {
         cell = motions.FRONT->NEXT;

         if ( motions.FRONT->NEXT->MOTN_REQUEST.REPETITIONS > 0 )
           if ( --motions.FRONT->NEXT->MOTN_REQUEST.REPETITIONS == 0 )
           {
             Dequeue( &cell, &motions );
             _Copy( cell, &motn_rply, sizeof( struct MOTN_RQST ) );
             motn_rply.MOTN_REQUEST.STD_REQUEST.MSG_SIZE =
                       sizeof( struct MOTN_RPLY );
             _Reply( (char *)&motn_rply, cell->REQUESTOR );
             Enqueue( cell, &freecells );
           }

         cell->MOTN_REQUEST.STD_REQUEST.MSG_SIZE =
                       sizeof( struct MOTN_RQST );
         _Reply( (char *)cell, requestor );
       }
       else
       {
         idle_rqst.STD_REQUEST.MSG_SIZE = sizeof( struct MOTN_RQST );
         _Reply( (char *)&idle_rqst, requestor );
       }
       break;
```

```
case MOTION_RQST:
  if ( request.MOTN_REQUEST.MODE != IDLE &&
       request.MOTN_REQUEST.MODE != MOVE )
    request.MOTN_REQUEST.REPETITIONS = 1;

  if ( Dequeue( &cell, &freecells ) )
    cell = (struct QCELL *)_Getvec( sizeof( struct QCELL ) );

  _Copy( &request, cell, sizeof( struct MOTN_RQST ) );
  cell->REQUESTOR = requestor;

  if ( request.MOTN_REQUEST.EXECUTE == IMMEDIATE )
    Push( cell, &motions );
  else
    Enqueue( cell, &motions );
  break;


case FLUSH_RQST:
  if ( !Empty( &motions ) )
  {
    freecells.REAR->NEXT = motions.FRONT->NEXT;
    freecells.REAR = motions.REAR;
    motions.REAR = motions.FRONT;
  }
  motn_rply.MOTN_REQUEST.MODE = IDLE;
  motn_rply.MOTN_REQUEST.STD_REQUEST.MSG_SIZE =
          sizeof( struct MOTN_RPLY );
  _Reply( (char *)&motn_rply, requestor );
  break;


case UNBLOCK_RQST:
  if ( Dequeue( &cell, &motions ) )
    motn_rply.MOTN_REQUEST.MODE = IDLE;
  else
  {
    if ( Empty( &motions ) )
      motn_rply.MOTN_REQUEST.MODE = IDLE;
    else
      _Copy( motions.FRONT->NEXT, &motn_rply,
             sizeof( struct MOTN_RQST ) );

    motn_rply.MOTN_REQUEST.STD_REQUEST.MSG_SIZE =
                  sizeof( struct MOTN_RPLY );
     Reply( (char *)&motn_rply, cell->REQUESTOR );
  }
```

```
  motn_rply.MOTN_REQUEST.STD_REQUEST.MSG_SIZE =
                    sizeof( struct MOTN_RPLY );
  _Reply( (char *)&motn_rply, requestor );
  break;


case UPDATE_RQST:
  if ( Empty( &motions ) )
    motn_rply.MOTN_REQUEST.MODE = IDLE;
  else
    _Copy( motions.FRONT->NEXT, &motn_rply,
                    sizeof( struct MOTN_RQST ) );
  motn_rply.MOTN_REQUEST.STD_REQUEST.MSG_SIZE =
                    sizeof( struct MOTN_RPLY );
  _Reply( (char *)&motn_rply, requestor );
  break;


case OPEN_REQUEST:
  {
    /* for now, we will not record connection */
    open_rply.STD_REPLY.MSG_SIZE = sizeof( struct OPEN_RPLY );
    open_rply.STD_REPLY.RESULT = OK;
    open_rply.UCB_SERVER = _My_id();
    open_rply.UCB_CONNECTION = 1;  /* arbitrary */
    open_rply.UCB_FLAGS = 0;
    open_rply.SIZE_UCB_XTRA = 0;

    _Reply( (char *)&open_rply, requestor );
    /* since connection not recorded, don't care if client died */
    break;
  }

case CLOSE_REQUEST:
 {
    /* for now, as connection not recorded,just reply */
    std_rply.MSG_SIZE = sizeof( struct STD_RPLY );
    std_rply.RESULT = OK;
     Reply( (char *)&std_rply, requestor );
    break;
 }

default:
   Breakpoint( "Aio_handler: bad msg_type. \n" );

  break;
```

```
    }

  }
};


Alt_courier()
{

  struct ALT_RQST alt_rqst;
  struct MOTN_RQST motn_rqst;
  uint_32 alt_serv_id;

  alt_serv_id = _Create( ALTSERV );
  alt_rqst.STD_REQUEST.MSG_TYPE = ALTER_RQST;
  alt_rqst.ERROR = ALTER_OK;

  for(;;)
  {
    motn_rqst.STD_REQUEST.MSG_SIZE = sizeof( struct MOTN_RQST );
    alt_rqst.STD_REQUEST.MSG_SIZE = sizeof( struct ALT_RQST );

    _Send( (char *)&alt_rqst, (char *)&motn_rqst, _Father_id() );

    motn_rqst.STD_REQUEST.MSG_SIZE = sizeof( struct MOTN_RQST );
    alt_rqst.STD_REQUEST.MSG_SIZE = sizeof( struct ALT_RQST );

    _Send( (char *)&motn_rqst, (char *)&alt_rqst, alt_serv_id );
  }

};


Init_q ( queue )

struct QUEUE *queue;
{
  queue->FRONT = queue->REAR =
                 (struct QCELL *)_Getvec( sizeof( struct QCELL ) );
  return( 0 );
};
```

```
Empty( queue )

struct QUEUE *queue;
{
  return( queue->FRONT == queue->REAR );
};


Push ( cell, queue )

struct QCELL *cell;
struct QUEUE *queue;
{
  if ( Empty( queue ) )
    queue->FRONT->NEXT = queue->REAR = cell;
  else
  {
    cell->NEXT = queue->FRONT->NEXT;
    queue->FRONT->NEXT = cell;
  }
  return( 0 );
};


Enqueue ( cell, queue )

struct QCELL *cell;
struct QUEUE *queue;
{
  queue->REAR->NEXT = cell;
  queue->REAR = cell;
  return( 0 );
};


Dequeue ( cell, queue )

struct QCELL **cell;
struct QUEUE *queue;
[
  if ( Empty( queue ) ) return( -1 );
  else
  {
    if ( (*cell = queue->FRONT->NEXT)  == queue->REAR )
      queue->REAR = queue->FRONT;
    else
```

```
      queue->FRONT->NEXT = queue->FRONT->NEXT->NEXT;
    return( 0 );
  }
};


typedef struct STD_RQST
  {
    uint_16              MSG_SIZE;
    int_16               MSG_TYPE;
  };


typedef struct STD_RPLY
  {
    uint_16              MSG_SIZE;
    uint_32              RESULT;
  };


typedef struct OPEN_RPLY
  {
    struct STD_RPLY      STD_REPLY;    /* RESULT is an error code */
    uint_32              UCB_SERVER;
    uint_32              UCB_CONNECTION;
    uint_32              UCB_FLAGS;
    uint_32              SIZE_UCB_XTRA;
  };


typedef struct UCB                     /* user connection block */
  {
    struct UCB           *NEXT_UCB;
    struct OPEN_RPLY     UCB_MAIN;
    char                 *UCB_XTRA;
  };


typedef struct AIO_DATA
  {
    struct STD_RQST      STD_REQUEST;
    int_32               DISPLACEMENTS[6];
    char                 *TEXT;
  };
```

```
struct DATA_RQST
  {
    struct STD_RQST              STD_REQUEST;
    int_16                       CONNECTION;
  };


struct TABLE
  {
    uint_32                      REQUESTOR;
    int_16                       VALID_DATA;
    int_16                       RPLY_PENDING;
  }


union AIO_MSG
  {
    struct STD_RQST              STD_REQUEST;
    struct DATA_RQST             DATA_REQUEST;
    struct AIO_DATA              FILTERED_DATA;
    struct OPEN_RQST             AIOH_OPEN;
    struct CLOSE_RQST            AIOH_CLOSE;
  };


struct GLOB_DATA
  {
    struct STD_RQST              STD_REQUEST;
    int_32                       OFFSETS[6];
    uint_32                      DISP_ID;
    uint_32                      PANIC_ID;
  };


struct TASK_TEMPLATE _Template_list[] =
  {
      { AIO, _Aio_filter_server, 1000, 1, 0 },
      { AIOHAND, Aio_handler, 1000, 3, 0 },
      { MIDPNTS, Midpoints, 1000, 4, 0 },
      { DISCOUR1, Disp_courier, 500, 5, 0 },
      { DISPLAY, Display, 800, 6, 0 },
      { 0, 0, 0, 0, 0 }
  };
```

```
struct INT_PAIR _Interrupt_list[] =
  {
    { 3, _RTI732_int },
    { 4, _Ptm_int },
    { 5, _Serial_int },
    { 0, 0 }
  };
```

TASK MIDPOINTS:

```
/*
 *  This task reads the AIO server and calculates the midpoint
 *  displacements. These are sent to the Aio_handler.
 */

Midpoints()
{
  union AIO_RQST aio_rqst;
  struct AD_RPLY ad_rply;
  struct AIO_DATA aio_data;
  struct STD_RQST std_rqst;
  struct STD_RPLY std_rply;
  struct UCB *ucb;

  ucb = _Open( "AIO: +rw", 0 );

  initaio( &aio_rqst, &ad_rply, GAIN, CHANNELS );
  aio_data.STD_REQUEST.MSG_TYPE = WRITE_REQUEST;


  for(;;)
  {
    Read_sensor( &aio_rqst, &ad_rply, ucb, aio_data.DISPLACEMENTS,
        CHANNELS );
    aio_data.STD_REQUEST.MSG_SIZE = sizeof( struct AIO_DATA );
    std_rply.MSG_SIZE = sizeof( struct STD_RPLY );
    _Send( (char *)&aio_data, (char *)&std_rply, _Father_id() );
  }

};
```

```
TASK AIOHANDLER:

/*
 *  This task is responsible for giving its clients valid sensor data.
 *  Sensor data is filtered by the Aio_filter_server, and multiplied by a
 *  calibration matrix in the task Midpoints(). It is then sent to this
 *  task. ( WRITE_REQUEST )
 *  This task treats the sensor data as a resource to be distributed
 *  amongst its clients.  For this reason it is structured as a server.
 *  The client must first open a connection, where it gets a connection
 *  number found in UCB_CONNECTION. All subsequent requests for data
 *  ( READ_REQUEST ) must contain this connection number in
 *  DATA_RQST.CONNECTION
 *  Aio_handler will reply to a client only with new sensor data.
 *  The client blocks until this data is available.
 */

Aio_handler()
{

  union AIO_MSG aio_msg;
  struct TABLE table[30];
  struct AIO_DATA aio_data;
  struct STD_RPLY std_rply;
  struct STD_RQST std_rqst;
  struct AIOH_INIT_REC aioh_init;
  struct OPEN_RPLY open_rply;
  uint_32 requestor;
  int_16 i, connect, tab_limit;



  std_rqst.MSG_SIZE = sizeof( struct STD_RQST );
  std_rqst.MSG_TYPE = INITIALIZE_SERVER;
  aioh_init.AIOHI_HDR.STANDARD.MSG_SIZE = sizeof( struct AIOH_INIT_REC );
  _Send( (char *)&std_rqst, (char *)&aioh_init, _Father_id() );
  if ( aioh_init.AIOHI_HDR.STANDARD.MSG_TYPE != AIOHIT_AIOH_INIT_REC )
      _Abort( "Aio_handler: bad init type. \n" );

  _Report_for_service( aioh_init.AIOHI_NAME, REPORT_FOR_SERVICE );

  _Create( MIDPNTS );
  tab_limit = 0;
```

```
for(;;)
{
  aio_msg.STD_REQUEST.MSG_SIZE = sizeof( union AIO_MSG );
  requestor = _Receive( (char *)&aio_msg, 0 );

  switch( aio_msg.STD_REQUEST.MSG_TYPE )
  {

    case READ_REQUEST:
      connect = aio_msg.DATA_REQUEST.CONNECTION;

      if ( connect >= tab_limit || table[connect].REQUESTOR !=
            requestor )
        _Breakpoint( "Aio_handler: connection error" );
      else
        if ( table[connect].VALID_DATA )
        {
          table[connect].RPLY_PENDING = TRUE;
        }
        else

        {
          table[connect].VALID_DATA = TRUE;
          aio_data.STD_REQUEST.MSG_SIZE = sizeof( struct AIO_DATA );
          _Reply( (char *)&aio_data, requestor );
        }
      break;


    case WRITE_REQUEST:
      _Copy( &aio_msg.FILTERED_DATA, &aio_data,
                      sizeof( struct AIO_DATA ) );
      for( i=0; i < tab_limit; i++ )
      {
        if ( table[i].RPLY_PENDING )
        {
          table[i].RPLY_PENDING = FALSE;
          aio_data.STD_REQUEST.MSG_SIZE = sizeof( struct AIO_DATA );
          _Reply( (char *)&aio_data, table[i].REQUESTOR );
        }
        else
        {
          table[i].VALID_DATA = FALSE;
        }
      }
```

```
        std_rply.MSG_SIZE = sizeof( struct STD_RPLY );
         Reply( (char *)&std_rply, requestor );
        break;


      case OPEN_REQUEST:
        table[tab_limit].REQUESTOR = requestor;
        table[tab_limit].VALID_DATA  = table[tab_limit].RPLY_PENDING =
              FALSE;
        open_rply.STD_REPLY.MSG_SIZE = sizeof( struct OPEN_RPLY );
        open_rply.STD_REPLY.RESULT = OK;
        open_rply.UCB_SERVER = _My_id();
        open_rply.UCB_CONNECTION = tab_limit++;
        open_rply.UCB_FLAGS = 0;
        open_rply.SIZE_UCB_XTRA = 0;
        _Reply( (char *)&open_rply, requestor );
        break;


      case CLOSE_REQUEST:
        connect = aio_msg.AIOH_CLOSE.CLOSE_CONNECTION;
        if ( connect >= tab_limit || table[connect].REQUESTOR !=
              requestor )
          _Breakpoint( "Aio_handler: connection error" );
        else
        {
          table[connect].VALID_DATA = table[connect].RPLY_PENDING = FALSE;
          std_rply.RESULT = OK;
        }
        std_rply.MSG_SIZE = sizeof( struct STD_RPLY );
        _Reply( (char *)&std_rply, requestor );
        break;

      default:
        _Breakpoint( "Aio_handler:  bad MSG_TYPE error" );
        break;

    }
  }

};
```

TASK DISPLAY:

```
/*
 *  Display.c - This program displays IRCC midpoint displacements in
 *  real-time.
 *  This program is designed to run only on a VT100 terminal.
 */

Display()

{
  char buffer[LENGTH+1];
  static char title[] = { 'X', 'Y', 'Z', 'x', 'y', 'z' };
  union AIO_RQST aio_rqst;
  struct AD_RPLY ad_rply;
  struct AIO_DATA aio_data;
  struct STD_RPLY std_rply;
  uint_32 requestor;
  int_16 i, num, print;
  int_32 new_offsets[CHANNELS], values[CHANNELS];


  _Selectoutput( _Open( "PO_TTY0: +w", 0 ) );

  _Printf( "\033[2J\033[4;24r" );

  for( i = 0; i < CHANNELS; i++ )
  {
    num = decode( &buffer, LENGTH, "%+ nac%nx", LENGTH-3, title[i], 3 );
    *(buffer + num) = '\0';
    _Putstr( buffer );
    _Flush();
  }

  _Printf( "\033[3;1f" );
  for( i = 0; i < 64; i++ )
    _Put( '-' );
  _Printf( "\033[4;1f\0337" );
  _Flush();

  for(;;)
  {
```

```
print = FALSE;

do
{
  aio_data.STD_REQUEST.MSG_SIZE = sizeof( aio_data );
  std_rply.MSG_SIZE = sizeof( std_rply );
  requestor = _Receive( (char *)&aio_data, 0 );
  _Reply( (char *)&std_rply, requestor );
  if ( aio_data.STD_REQUEST.MSG_TYPE == KBD ) print = TRUE;
}
while( aio_data.STD_REQUEST.MSG_TYPE == KBD );

if ( print )
{
  aio_data.STD_REQUEST.MSG_TYPE = PRINT_VALUES;
  aio_data.TEXT = "";
}

switch ( aio_data.STD_REQUEST.MSG_TYPE )
{

  case SHOW_VALUES:

    _Printf( "\033[2;1f" );
    Vsub( values, aio_data.DISPLACEMENTS, new_offsets, CHANNELS );
    Print_data( values );
    _Flush();
    break;


  case PRINT_VALUES:

    _Printf( "\0338" );
    _Printf( "%s", aio_data.TEXT );
    Print_data( aio_data.DISPLACEMENTS );
    _Printf( "\n\0337" );
    _Flush();
    break;


  case NEW_OFFSETS:

    for( i=0; i < CHANNELS; i++ )
      new_offsets[i] = aio_data.DISPLACEMENTS[i];
    break;
```

```
      default:

        _Breakpoint( "Display : BAD AIO_DATA.STD_REQUEST.MSG_TYPE" );
        break;

    }
  }
};


Print_data( values, offset )

int_32 values[];
{

  int_16 i, num;
  char buffer[LENGTH+1];
  float p;


  for( i = 0; i < CHANNELS; i++ )
  {
    p = (float)values[i] * AIO_SCALE;
    num = decode( &buffer, LENGTH, "%+ n.4f", LENGTH, p );
    *(buffer + num) = '\0';
    _Putstr( buffer );
  }

};
```

```
TASK DISP_COURIER:

Disp_courier()
{

  struct AIO_DATA aio_data;
  struct STD_RQST std_rqst;
  struct DATA_RQST data_rqst;
  struct STD_RPLY std_rply;
  struct UCB *aioh_ucb;
  struct GLOB_DATA glob_data;

/*  Get display task id from father  */

  glob_data.STD_REQUEST.MSG_SIZE = sizeof( struct GLOB_DATA );
  std_rply.MSG_SIZE = sizeof( struct STD_RPLY );
  _Receive( (char *)&glob_data, _Father_id() );
  _Reply( (char *)&std_rply, _Father_id() );

/*  Open connection to Aio_handler  */

  aioh_ucb = _Open( "AIOHAND:", 0 );

  data_rqst.STD_REQUEST.MSG_SIZE = sizeof( struct DATA_RQST );
  data_rqst.STD_REQUEST.MSG_TYPE = READ_REQUEST;
  data_rqst.CONNECTION = aioh_ucb->UCB_MAIN.UCB_CONNECTION;
  aio_data.STD_REQUEST.MSG_SIZE = sizeof( struct AIO_DATA );
  std_rply.MSG_SIZE = sizeof( struct STD_RPLY );

  for(;;)
  {
    _Send( (char *)&data_rqst, (char *)&aio_data,
           aioh_ucb->UCB_MAIN.UCB_SERVER );
    aio_data.STD_REQUEST.MSG_TYPE = SHOW_VALUES;
    _Send( (char *)&aio_data, (char *)&std_rply, glob_data.DISP_ID );
  }

};
}

/*
 *  reads Aio_filter_server and converts to displacements
 */

Read_sensor( aio_rqst, ad_rply, ucb, midpnts, channels )
```

```
union AIO_RQST *aio_rqst;
struct AD_RPLY *ad_rply;
struct UCB *ucb;
int_32 midpnts[];
uint_32 channels;
{
     int_32 volts[CHANNELS], i;

     _Send( (char *)aio_rqst, (char *)ad_rply, ucb->UCB_MAIN.UCB_SERVER );

     for( i = 0; i < channels; i++ )
     {
          /* 12-bit A/D value is returned in MSB's and must be shifted to
the right 4 bits */
          volts[i] = (int_16)ad_rply->AD_VALUES[i] >> 4;
     }

/*
 *  converts IRCC voltages to midpoint displacements
 *  These constants are taken from a calibration matrix provided by
 *  C.S. Draper Labs.
 */

   midpnts[0] =  8625 * volts[0] - 7842 * volts[2];
   midpnts[1] =  4907 * volts[0] - 8825 * volts[1] + 4462 * volts[2];
   midpnts[2] = -4004 * volts[3] - 4277 * volts[4] - 4394 * volts[5];
   midpnts[3] = -267 * volts[3] + 293 * volts[5];
   midpnts[4] = -152 * volts[3] + 324 * volts[4] - 166 * volts[5];
   midpnts[5] = -186 * volts[0] - 165 * volts[1] - 169 * volts[2];

};
```